Failure-tolerant parallel programming and its supporting system architecture

by K. H. KIM
University of Southern California
Los Angeles, California

and

by C. V. RAMAMOORTHY
University of California
Berkeley, California

ABSTRACT

The state-of-art in software validation as well as the continuing growth of the size and complexity of software subsystems, makes extra costs paid for software error tolerance more than justified. A program in which software redundancy is incorporated i.e. a program in which procedures for run-time validation and recovery are explicitly specified, is generally called a failure-tolerant program. One problem in failure-tolerant programming, which could be particularly serious in real-time computing environments, is the program execution time increased due to incorporation of validation and recovery procedures. This paper introduces an approach to the solution, called the failure-tolerant parallel programming. The essence of this approach is to maximally overlap main-stream computation with redundant computation oriented for validation and recovery. Subsequently, a model system architecture tailored for efficient execution of failure-tolerant parallel programs is described. It is of highly general and modular nature and contains a novel memory subsystem named the duplex memory. Directions of further researches on program structuring and expansion of the model architecture are also indicated.

INTRODUCTION

Computing system reliability is a function of both *hardware reliability* and *software reliability*. Hardware failures occur due to physical component *faults* (i.e., material characteristics) or *design errors*. The former source has been dominating the latter in significance. Recent advances in hardware component technology have substantially reduced the occurrences of hardware faults, thus greatly improving hardware reliability. On the other hand, all software failures are due to design errors. As the size and complexity of software subsystems grows steadily, software reliability has become a very serious problem and an increasingly important factor determining the overall system reliability.

Complete validation, which assures the absolute correctness of a program through verification of its complete behavioral characteristics, still remains to be infeasible with sizable programs.[2,12] On the other hand, more popular and pragmatic approaches aiming at *partial validation* with high cost-effectiveness via testing cannot, by their nature, insure the absence of errors in the program.[10,13,16,20,21] The apparent consequence is the current practice in which errors remain to exist in large programs put into operation. It is also this practice that makes *software error tolerance* an important objective besides complete removal of software errors at the design stage.

The concept of *failure-tolerant computing* i.e., reliable computing despite the presence of system component failures, was born in the very early days of electronic computing.[18,26] Since then, *hardware fault tolerance* has been a main subject of extensive investigation.[1,3,7,25] *Redundancy* is a fundamental vehicle in realizing failure-tolerant computing. A majority of previous studies have been centered around the use of hardware redundancy and in contrast, very little studies were made on the use of software redundancy. A restricted amount of software redundancy has been exploited in the form of *rollback and recovery* defined as follows. Let *state vector* refer to a snapshot of the contents of all the variables of the program in execution. Rollback and recovery is a technique of depositing state vectors at several stages in the middle of program execution and in case of a system failure, resetting the system state by using an old state vector and restarting the execution from that stage. However, the way failure detection, state vector saving and

recovery operations are designed and specified has been mostly ad hoc and heuristic. It was only in recent years that studies were made on systematic and cost-effective implementation of a rollback and recovery scheme.[4,5,6,15,19,23,24]

In case of hardware faults, rollback (possibly combined with system reconfiguration using redundant hardware components) and re-execution with the same program will suffice to get over the situation. However, such an approach does not help in case of software failures. From the very nature of software errors, software error tolerance requires more extensive exploitation of redundancy, particularly software redundancy which is essentially a design redundancy. The method of structuring programs in which software redundancy is explicitly incorporated, is generally called *failure-tolerant programming*. It was in recent years that software error tolerance became a subject of serious studies and research was initiated toward the development of structured failure-tolerant programming techniques.[8,11,14,17,22,27]

In the next section, a brief overview of those recent significant contributions is given. Then some desirable directions of extending the state-of-art in failure-tolerant programming, which, we believe, are significant in real-time computing environments, are pointed out. The following section introduces a new approach to failure-tolerant programming (termed failure-tolerant parallel programming) devised to be a desirable extension of the state-of-art, and discusses the requirements on the system architecture oriented for efficient execution of failure-tolerant parallel programs. The following section describes an architecture developed to satisfy the requirements discussed in the preceding section. Finally, areas of extension and further research are discussed and then this paper is concluded.

BACKGROUND

Recent research on failure-tolerant programming and software error tolerance made significant contributions in the following aspects:

First, the notion of a *failure-tolerant program* was solidified. A failure-tolerant program is essentially a self-checking and recovering program. More specifically, a failure-tolerant program contains specifications of the procedures of validating intermediate results at various stages during execution and recovering when an abnormal condition is detected as a result of the check. Thus it consists of two types of program-segments: (1) *object segments* specifying application-oriented computations, and (2) *validation and recovery* (*VR-*) *segments*, each associated with a certain object segment and specifying the procedures of validating the results produced by the associated object segment and recovering in case of incorrect results. Within a failure-tolerant program, powerful facilities for validation and recovery can be incorporated in a systematic manner to any desirable extent. Here "recovery" implies not just the repetition of the execution with the same object segment (which may have failed the validation-test due to the hardware faults or the errors contained in it) but rather the provision of a set of "alternate" object segments and trials with one after another until a certain alternate object segment passes the validation test. If all the alternatives fail, then either the program cannot be successfully completed or a more global recovery action is incurred, provided the failed object segments are nested in another object segment and the latter is associated with a VR-segment.

Importance of good structure in failure-tolerant programs is evident, since structuring a failure-tolerant program by introducing VR-segments into a conventional program containing only object segments is accompanied by an increase in program size and complexity. Recognizing this importance, Randell's group at the University of Newcastle upon Tyne, England developed an experimental scheme called *recovery block structuring* by which validation and recovery functions can be embedded, in a well-structured form, inside each block in programs written in block-structured languages like ALGOL.[11,22] To give some flavor to this structuring scheme, the structure of the *recovery block* (i.e., the failure-tolerant block) is depicted in Figure 1.

In the diagram, double vertical lines define the bodies (i.e. scopes) of recovery blocks, while single vertical lines define the bodies of primary or alternate object blocks. The *primary object block* corresponds exactly to the block of the equivalent conventional

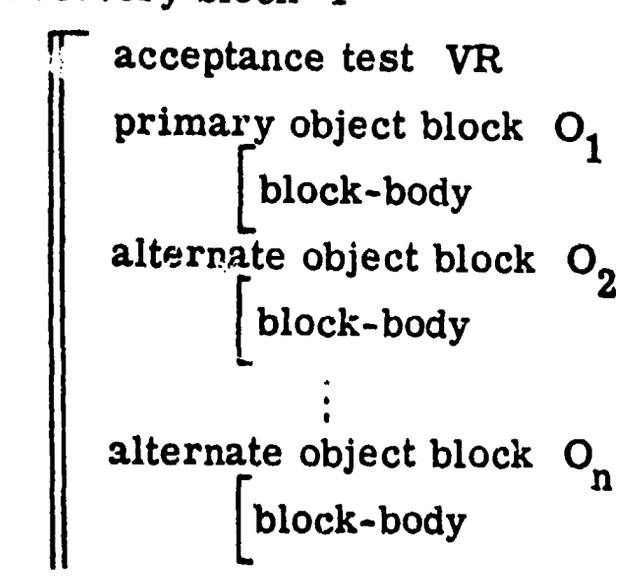

Figure 1—A structural model of the recovery block developed in References 14 and 22

(ALGOL-like) program and is a kind of an object segment. The *acceptance test* is executed on exit from an object block to confirm that the object block has performed acceptably. If confirmed, the control exits from the recovery block. Thus the acceptance test is a kind of a VR-segment. If the result produced by an object block is determined to be unacceptable, the next *alternate object block* is entered and is required to perform the objective operation in a different way or to perform some alternate action acceptable to the program as a whole. The acceptance test is then repeated.

The following aspects of recovery block structuring are rather fundamental and may be found, possibly in different formats, in any structured approach to the failure-tolerant programming.

(1) The primary or alternate object blocks can contain, nested within themselves, further recovery blocks.
(2) When an alternate object block needs to be entered after the result of the preceding object block fails the acceptance test, the system state must be restored to the one current just before entry to the primary object block.
(3) Execution of the acceptance test upon exit from an object block generally requires the reference to both the original values and the modified values of the variables non-local to the object block.
(4) It is not necessary that every block in a block-structured failure-tolerant program be a recovery block.

Second, a technical basis was established for reducing the overhead involved in saving a state vector on entry to each object segment and resetting the system state by using a saved state vector during recovery. The overhead exists in two forms. One is the processor time spent for those activities and the other is the store space occupied by saved state vectors. A useful property which can be advantageously exploited for overhead reduction is that the variables local to the object segment are irrelevant to the recovery and in many cases, only a few of the non-local variables are modified by the object segment.

Based on this, Randell's group developed a scheme for state vector saving and system state resetting, called a *recursive cache mechanism*, to support execution of programs structured by the scheme of recovery block structuring.[11] The essence of this scheme is to save the original value of each non-local variable together with its name (i.e., its logical address) right before the variable is modified for the first time in a new object block. Thus state vectors are saved in compact forms. It is apparently necessary to detect, at run-time, whether an assignment to a non-local variable is the first to have been made to that variable within the current block. This capability is provided by the *flag* attached to each non-local variable. Again, to give some flavor to this mechanism, an example of the recursive cache is shown in Figure 2.

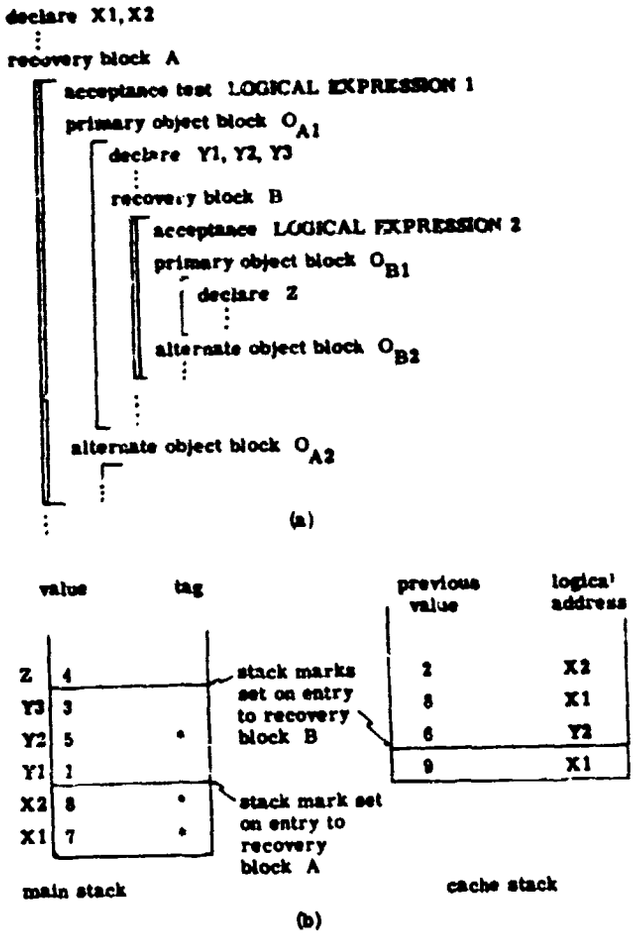

Figure 2—(a) A program structured by the recovery block structuring (b) A snapshot of the recursive cache during execution of (a)

Figure 2(a) shows a failure-tolerant program structured by the recovery block structuring scheme. Figure 2(b) shows a snapshot of the recursive cache taken when primary object block O_{B1} is in the middle of its execution. There are two stacks, the *main stack* and the *cache stack*. The cache stack is also divided into regions, one for each nested recovery block in "active" state. The top region of the cache stack in Figure 2(b) contains previous values of non-local variables together with their names i.e., Y2, X1, X2, which have been modified by execution of the current object block O_{B1}. The flags attached to those non-local variables in the main stack are set accordingly. Similarly, the bottom region of the cache stack contains the previous value of non-local variable X1 which had been modified by execution of object block O_{A1} before O_{B1} was entered. If the result produced by execution of O_{B1} fails the acceptance test (LOGICAL EXPRESSION 2), then the top region of the cache stack can be used to reset the

content of the main stack to the one current before entry to recovery block B. If it passes the test, the top region is merged into the bottom region of the cache stack so that the result will contain previous values of those variables which are non-local to object block O_{A1} and have been modified since O_{A1} was entered. Thus the result will be a single region containing (9, X1) and (2, X2). Flags in the main stack are also adjusted such that only flags of X1 and X2 be set. Therefore, the combination of the main and cache stacks contain information on the basis of which several old state vectors can be reconstructed.

This is perhaps an oversimplified account of recent developments. Yet it is intended to provide all the essential backgrounds for clarifying main departures of our works presented in the rest of this paper. For more information on the schemes described in this section, readers are of course referred to their original reports.[11,12]

DESIRABLE EXTENSIONS OF THE STATE-OF-ART IN FAILURE-TOLERANT PROGRAMMING

On the basis of recent works in failure-tolerant programming, particularly those introduced in the preceding section, various extensions can be clearly envisioned. Among many desirable extensions, the following ones are considered to be of great significance.

First, one problem in failure-tolerant programming, which could be particularly serious in real-time computing environments, is the program execution time increase due to incorporation of VR-segments. In most of the previous approaches including the recovery block structuring and recursive cache schemes introduced in the preceding section, validation and saving of state vectors fully contribute to the increase of the program execution time. Consequently, when any non-trivial validation is employed or large numbers of non-local variables are modified during execution of object segments, the program execution time, even in the case of normal failure-free operation, could very well exceed the tolerable limit in real-time applications. It is indeed expected that the VR-segment will be frequently a quite complex program-segment. Even in the recovery block structuring scheme in which only a restricted form of a VR-segment i.e. a logical expression is allowed for the sake of reducing error-proneness of the VR-segment, a provision has been made to allow procedure calls within acceptance tests (i.e., logical expressions).[11]

Second, the rationale underlying the restriction of the acceptance test to a logical expression is considered a perfectly legitimate one. Yet the logical expression is considered an excessively restrictive form of a VR-segment in many environments. For instance, it may be desirable to explicitly specify in the VR-segment which alternate object segment, among multiple alternates, is to be tried next in each case of recovery, rather than always letting the system select the next alternate object segment randomly or in the order alternates are located in the program text. It may also be desirable to immediately revert to a global recovery if a certain erroneous condition is detected by execution of a VR-segment, instead of retrying with an alternate. That is, allowance of more flexible structu in failure-tolerant programs may be desirable. Furthermore, our approach toward the first desirable extension mentioned above i.e., execution of VR-segments with minimal increase in the overall program execution time, favors more flexibility in structuring failure-tolerant programs. This will become evident in the next section.

CONCEPT OF FAILURE-TOLERANT PARALLEL PROGRAMMING AND REQUIREMENTS ON THE SUPPORTING SYSTEM ARCHITECTURE

Our main concern in this paper is with the first desirable extension mentioned in the preceding section, that is, incorporation of VR-segments with minimal increase of program execution time.

The fundamental approach we have adopted is to maximally overlap execution of object segments with execution of VR-segments. Since the VR-segment specifies manipulation on the results produced by its associated object segment, dependency of the former for its initiation on the completion of the latter is inherent. However, it is possible to execute the VR-segment associated with an object segment concurrently with the successor object segment(s). Figure 3 illustrates this concept. There VR-segment VR_2 can be initiated only after completion of the correspondent object segment O_2 and VR-segment VR_1, but it may be executed concurrently with object segments O_3, O_4, etc. In an ideal situation where execution of VR-segments is fully overlapped with execution of object segments, the amount of increase in program execution time will be the time required for execution of the last VR-segment (e.g., VR_n in Figure 3) since it is the only VR-segment which cannot be executed in overlap with object segments.

A failure-tolerant program in which computational parallelism, especially parallelism between application-oriented computations and redundant computations for validation and recovery, is explicitly indicated, is called a *failure-tolerant parallel program*. That is, the main type of parallelism which characterizes a failure-tolerant parallel program is the one existent between object segments and VR-segments.

This approach requires a new method of structuring a failure-tolerant program. The major departure of a newly required structuring method from the previously developed ones is in specification of the control structure among program-segments. In addition to inherent dependency of VR-segments on their correspondent object segments, dependency of object segments on VR-segments may also be specified in a failure-tolerant

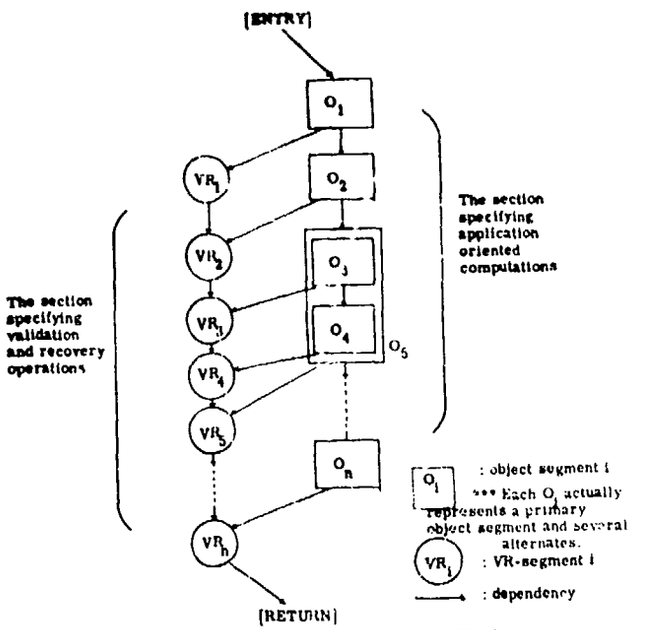

Figure 3—A simple example of a failure-tolerant parallel program

parallel program. An example of such a situation is when a certain object segment specifies a critical operation such as ordering an emergency action, erasing a secret file, etc. In such a case it is desirable to suspend the execution of the object segment until all the VR-segments corresponding to its predecessor object segments have been verified. Thus critical operations can be controlled to occur reliably. This clearly favors, if not requires, the second desirable extension discussed in the preceding section i.e., more flexibility in structuring VR-segments and overall failure-tolerant programs. Yet such flexibility can be obtained without sacrificing most desirable structuring principles (or strategies) underlying the developed structuring schemes, including the recovery block structuring scheme. This aspect will be discussed later.

On the other hand, the failure-tolerant parallel programming imposes the following requirements on the supporting system architecture i.e., the architecture of a system capable of efficient execution of failure-tolerant parallel programs.

First and the most obvious of all, the system must contain at least two processors, one for execution of object segments, called the *object processor* and the other for execution of VR-segments, called the *VR-processor*.

Second, the state vector at the completion of an object segment is an input data not only to the successor object segment but also to the associated VR-segment. In the rest of this paper, a state vector refers to a snapshot of all the variables appearing in object segments, not including local variables defined in VR-segments, taken at one moment. Furthermore, the VR-segment requires the input state vector undestroyed until it no longer needs to examine it, while the successor object segment, by nature, continuously changes it into the up-to-date one. It is thus necessary to create a "copy" of the state vector current at the end of the execution of an object segment for exclusive use by the VR-processor executing the associated VR-segment. Here the process of creating a copy must cause no or little delay in executing object segments. The store space containing copies of state vectors must also be minimized. If either execution time or store space exceeds the tolerable limit, the objective of failure-tolerant parallel programming is defeated. This requires a new store management scheme substantially different from the previously developed ones. This point will become clearer when we propose one suitable scheme in the next section.

Third, if execution of VR-segments lags much behind execution of object segments, there will be accumulated a large number of unprocessed copies of state vectors. Thus when all the available store space runs out, the object processor must be suspended until the VR-processor catches up. In order to avoid this undesirable situation, execution of VR-segments must be speedy.

A MODEL ARCHITECTURE SUPPORTING FAILURE-TOLERANT PARALLEL PROGRAMMING

In this section we describe a system architecture oriented for efficient execution of failure-tolerant parallel programs. We call it a model architecture since it is of highly general nature and thus is specified at an abstract level. Yet it is expected that elaboration of the architecture into a specific working system will encounter no new logical problems of fundamental nature.

The store management scheme

Potential power of failure-tolerant parallel programs cannot be realized without accompanying the additional cost of the supporting system architecture. The additional cost is paid in the forms of both processor redundancy and store redundancy. Since the object processor runs concurrently with the VR-processor, memory conflicts must be carefully avoided. This rules out the feasibility of having a single state vector or its portion shared by both processors. Thus each processor owns a region of the store during execution of a failure-tolerant parallel program.

The region of the store used by the object processor is called the *main working store*, while the region of

the store used by the VR-processor is called the *VR-store*. The VR-store contains copies of state vectors including the up-to-date one plus possibly more than one old one. This, of course, does not mean that the VR-store contains complete duplicates of several state vectors. Let S_i, S_{i+1}, ------, S_{i+j} denote a (chronological) sequence of state vectors that can be reconstructed from the content of the VR-store. Then the VR-store actually contains one complete copy of S_i (i.e., the oldest reconstructable one) and only the differences between pairs of adjacent state vectors in the sequence i.e. $S_{i+1}-S_i$, $S_{i+2}-S_{i+1}$, -------, $S_{i+j}-S_{i+j-1}$.

More specifically, as the object processor executes each object segment, it produces the *execution image* which consists of the values of variables assigned during execution of that object segment. If a variable is assigned several times during execution of the object segment, only the latest assigned value is contained in the execution image. The execution image produced on completion of an object segment represents the difference between the state vector current right before entry to the object segment and the up-to-date state vector (i.e. the one current on completion of the object segment). Each execution image is stored in a segment of the VR-store called a *VR-store-segment*. Each execution image is examined by the VR-processor to determine the acceptability of the result produced by execution of the object segment.

The execution image of an object segment consists of values of both local variables and non-local variables. Thus each VR-store-segment consists of two sections, one for local variables and the other for non-local variables. On entry to each object segment, memory space is allocated for the segment of the main working store containing the set of local variables defined within the object segment. At that time, the same size of memory space is also allocated for the section of the VR-store-segment containing (a copy of) the set of local variables. However, store space for non-local variables is not entirely duplicated. Instead, the section of the execution image containing non-local variables is written in the form of a table in which each entry consists of the logical address and the new value of a non-local variable. The idea is to take advantage of the useful property that in many cases, only a few of the non-local variables are modified by an object segment, while the total number of non-local variables defined may be very large. Thus the table representation leads to a highly compact form of the non-local variable section of the execution image.

As a simple example, consider a block-structured program augmented with VR-segments in Figure 4(a). Figure 4(b) shows snapshots of the main working store i.e. the stack used during execution of object segments in the program. In Figure 4(c) VR-store-segment 1 (or 2, 3, 4, 5) is used to contain the execution image of object segment O_1 (or O_2, O_3, O_4, O_5). Each VR-store-segment except VR-store-segment 1

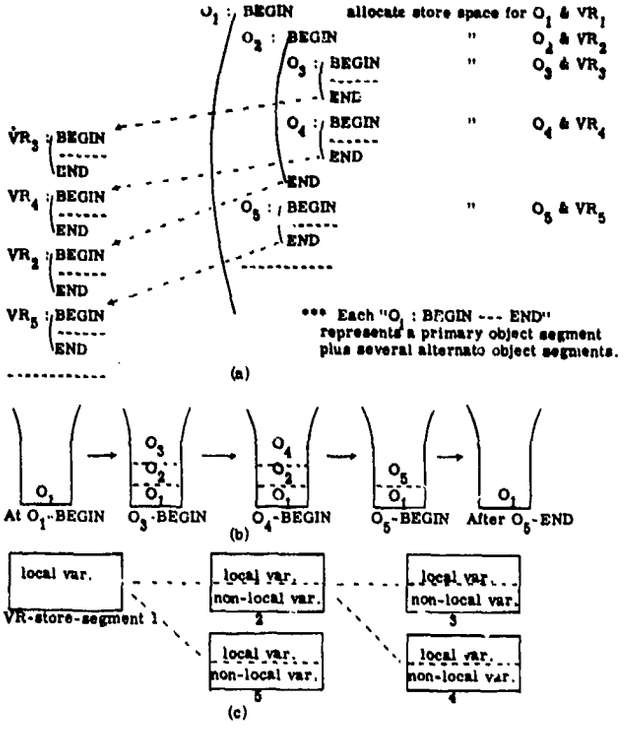

Figure 4—(a) A failure-tolerant parallel program (b) Snapshots of the mail working store during execution (c) A snapshot of the VR-store during execution

consists of two sections, one for local variables and the other for a table holding newly assigned values of non-local variables and their logical addresses. Each VR-store-segment is created on entry of the object processor to the correspondent object segment.

At the beginning of VR-segment VR_3, VR-store-segment 3 contains the execution image of object segment O_3 and the object processor has probably entered into O_4. The execution image in VR-store-segment 3 is examined by execution of VR_3. When it has been verified or judged to be acceptable, the local variable section is discarded and the non-local variable section is merged into VR-store-segment 2. If two different values of the same variable were contained in both VR-store-segments 3 and 2, the value in VR-store-segment 2 is the older one and replaced by the value in VR-store-segment 3. At the beginning of VR_4, VR-store-segment 3 does not exist. Then it is no longer possible to reconstruct the state vector which was current right before entry to O_3. There will be no need to reconstruct that state vector since VR_3 has been successfully completed.

Similarly, upon successful completion of VR_2, the values of non-local variables in VR-store-segment 2 are absorbed into VR-store-segment 1. At the beginning

of VR_3, VR-store-segments 2, 3 and 4 do not exist. Then the state vector which was current at the initiation of O_2 can no longer be reconstructed.

Here one subtle problem is noteworthy. Consider a simple program in Figure 5(a). Since there are two object segments (thus two VR-segments), there are two VR-store-segments (1 and 3). The body of object segment 1 consists of the upper section, O_3 and the lower section. During execution of the upper section, the execution image will be stored in VR-store-segment 1. At the completion of O_3, VR-store-segment 3 contains the execution image and the VR-processor may start examining it. Then the object processor executes the lower section of O_1 and it stores the execution image into VR-store-segment 1 which already contains the execution image of the upper section. Some values in VR-store-segment 1 are now the ones assigned by execution of the upper section, while others are the ones assigned by execution of the lower section. When the VR-processor has successfully completed VR_3 and needs to merge the verified execution image into VR-store-segment 1, it is not possible to tell for each variable whether the value in VR-store-segment 1 is the older one than the one in VR-store-segment 3.

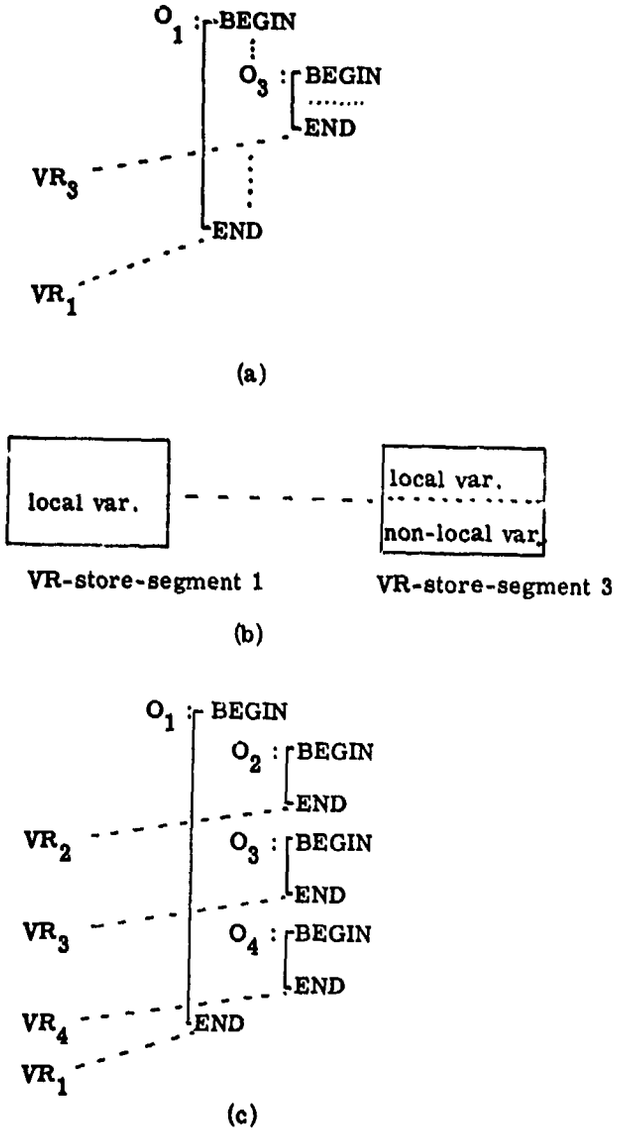

Figure 5—(a) An unconstrained program (b) A snapshot of the VR-store (c) An equivalent program satisfying the constraint

This problem can be resolved by imposing an additional constraint on program structuring. That is, the upper section and the lower section of O_1 in Figure 5(a) must be changed into O_2 and O_4, respectively, nested within O_1. This results in Figure 5(c). Then there will be additional VR-store-segments (2 and 4) created during execution, and the above problem disappears. Therefore, the constraint is that either each object segment contains no other object segments nested in it or its entire body must be composed of other object segments nested in it. This constraint is incorporated only for secure allocation of the VR-store. Thus the programmer is not required to prepare new VR-segments VR_2 and VR_4.

If there are no explicit VR-segments associated with some object segments, the system will insert dummy VR-segments. Or a variation of the above solution is to make the system responsible for restructuring unconstrained programs (e.g., Figure 5(a)) into the ones satisfying the constraint (e.g., Figure 5(c)), rather than imposing the constraint on the programmer. In any case, the solution does not appear to be a costly one.

The model system structure

The store management scheme described above will work (almost) perfectly if execution images can be created in the VR-store without causing any delay in execution of object segments. As far as the local variable section of the VR-store is concerned, this condition can be met without employing unconventional hardware components. Since the same size of the store-segment for local variables is allocated in both the main working store and the VR-store, the only requirement is to set proper base addresses for both store-segments and, for each assignment, write the same value in two locations of the same relative address, one in the main working store and the other in the VR-store.

However, the situation is different in creating the non-local variable section of the execution image. As described before, the non-local variable section is written in the form of a table. Thus whenever a non-local variable is assigned a new value during execution of an object segment, the value is written into the correspondent location in the main working store and at the same time, the value together with the logical address is written into an entry of a table in the VR-store.

If the assignment to a non-local variable is the first to have been made to that variable within the current object segment, then the new value and the logical address of that variable are written into the next available (empty) entry of the table in the VR-store. If it is not the first, there are two choices. One is to write in the same manner as above i.e. write into a new entry of the table. The other is to locate the entry of the table containing the previous value and the logical address of the non-local variable and then replace the previous value with the newly assigned one. After all, only the latest assigned values of variables need to be contained in the execution image. The former approach leads to the larger table-size than the one resulting from the latter approach. In addition, when or before the VR-processor executes the VR-segment to validate the table, it must discard the entry (of the table) containing the older value of each variable for which there is another entry containing the later assigned value. The latter approach leads to the compact table but it requires a special hardware support in order not to degrade the performance of the object processor. The special hardware requirement can be met by incorporation of the *content-addressable* (*i.e., associative*) *memory module* whose access time closely matches the one of the location-addressed memory module used in the system. In addition, incorporation of such a memory module significantly enhances the performance of the VR-processor in executing VR-segments. In view of the decreasing trend of the hardware cost and the performance advantage, the latter approach is considered favorable.

It is also necessary to attach a tag to the logical address of each variable which indicates whether the variable is a local one or a non-local one.

The structure of the model system in which all the above decisions are reflected, is depicted in Figure 6. The model system contains multiple central processors (CP's) and the memory subsystem named the *duplex memory*.

Each CP may function as an object processor, a VR-processor, a supervisory processor or a spare at one time. Employment of general purpose CP's is motivated mainly by the consideration of the flexibility in system reconfiguration. Yet this is not an absolute necessity and can be compromised for employment of processors fixed for a specific function if other factors such as cost and performance dictate so.

The duplex memory contains two types of memory modules, location-addressed memory modules and content-addressable memory modules. Location-addressed memory modules are further divided into two sets. One set of modules provides the main working store for the object processor. The other set, together with the set of content-addressable memory modules, provides the VR-store. Thus the local variable section of each execution image is contained in location-addressed memory modules, while the non-local variable section is

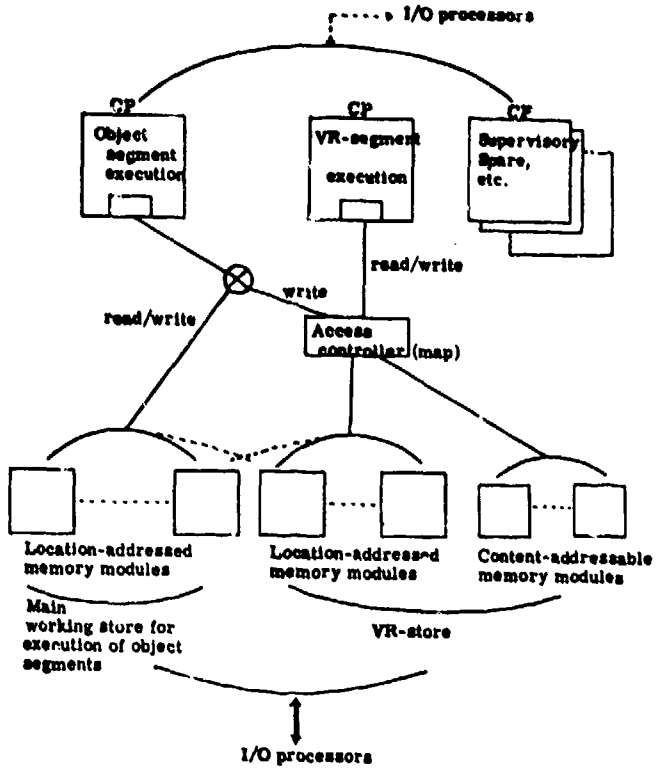

Figure 6—A system architecture based on multiple CP's and the duplex memory

contained in content-addressable memory modules. That is, whenever the object processor issues a "write" command, the value is written into two locations simultaneously, one in the main working store and the other in the VR-store. The latter location is in a location-addressed memory module if the tag attached to its logical address indicates "local" or in a content-addressable memory module if the tag indicates "non-local."

The VR-processor never accesses the main working store except during recovery. The object processor never reads from the VR-store. It is a natural property of this duplex memory that the partition of location-addressed memory modules consisting of two disjoint sets, (one providing the main working store and the other providing the local variable section of the VR-store) may change dynamically.

There is another important reason for employing content-addressable memory modules. Execution of a VR-segment generally involves not only examination of the correspondent execution image but also references to other *ancestor* VR-store-segments i.e., ones created prior to the VR-store-segment containing the correspondent execution image. For instance, when the VR-processor executes VR-segment VR_4, in Figure 4, it examines the content of the correspondent VR-

store-segment 4 and it may access ancestor VR-store-segments 1 and 2. References to ancestor VR-store-segments are for obtaining the previous values of non-local variables (i.e., variables non-local to the object segment associated with the VR-segment currently in execution).

It is frequently required for each non-local variable to obtain the latest assigned value among all of its values contained in ancestor VR-store-segments. The desired value may exist in the local variable section of an ancestor VR-store-segment. In this case, the variable is defined within the object segment which created the ancestor VR-store-segment. The desired location (which is in a location-addressed memory module) is exactly the one to which the logical address of the variable is mapped, and thus it can be directly accessed.

On the other hand, the desired value may exist in the non-local variable section of an ancestor VR-store-segment. For instance, assume in Figure 4 that variable Z is defined in object segment O_1, initialized with "100," assigned "200" by execution of O_1 and assigned "300" by execution of O_2. Let us also assume that the VR-processor is currently executing VR_4 and it needs to obtain the value of Z assigned the latest before the object processor entered into O_2. The desired value is "200" and it is contained in the non-local variable section of VR-store-segment 2 since VR_2 has already been successfully completed. The non-local variable section of VR-store-segment 2 needs to be searched in order to get the desired value "200." The problem here is a little complex since if Z has not been assigned a value during execution of O_1, the desired value is "100" contained in the local variable section of VR-store-segment 1. It is not known in advance which VR-store-segment, between 1 and 2, contains the desired value. It is thus necessary to check VR-store-segment 2 first and if its non-local variable section does not contain Z, then the desired value is read from the location, in the local variable section of VR-store-segment 1, to which the logical address of Z is mapped.

In general, it is necessary to search ancestor VR-store-segments in the increasing order of their ages, until either an ancestor containing the variable in its non-local variable section is found or the ancestor addressed by a portion of the logical addresses of the variable is reached. Here employment of content-addressable memory modules for non-local variable sections of execution images is the key to the high speed of search. It enables simultaneous examination of non-local variable sections of all the "ancestor VR-store-segments" which are "descendants" of the one containing the variable in its local variable section. When the VR-processor issues a command for fetching the latest assigned value of a non-local variable among its values contained in ancestor VR-store-segments, one location-addressed memory module and possibly several content-addressable memory modules are simultaneously accessed. The location-addressed memory module is the one addressed by a portion of the logical address of the variable. The content-addressable memory modules are the ones containing the non-local variable sections of those ancestor VR-store-segments which are descendants of the one whose local variable section is mapped to the above location-addressed memory module. Selection of the latest assigned value among all the values of the variable retrieved from those memory modules is also a function of the duplex memory. With this duplex memory, the VR-processor can read or update any value contained in the VR-store with the amount of time close to one content-addressable memory cycle.

Resetting the system state when an execution image is evaluated to be unacceptable, is also speedy with this duplex memory. It is because the resetting process involves basically fetching the previous value of each non-local variable which has been assigned another value since the object processor entered into the object segment whose execution image turned out to be unacceptable, and then storing it into the location of the variable in the main working store. Each non-local variable which needs to be restored in the main working store to its previous value is identified by examining the non-local variable sections of the execution images produced since the object processor entered into the object segment whose execution image was rejected.

Local variable sections of VR-store-segments are mapped to location-addressed memory modules in the same manner as the main working store is mapped to location-addressed memory modules. For some VR-segments, each of their non-local variable sections may be mapped to an independent content-addressable memory module. For others, their non-local variable sections may co-exist inside the common content-addressable memory module, provided some form of an identification code is assigned to each section. The high speed requirement limits the size of each content-addressable memory module to a small one. It may sometimes be necessary to use more than one content-addressable memory module to hold the non-local variable section of a VR-store-segment.

It is believed that this architecture satisfactorily meets all the requirements mentioned before. Creating a copy of a state vector in this system does not incur any delay in processing object segments. The memory interference between the CP's executing object segments and VR-segments is absent or negligible. Employment of multiple content-addressable memory modules is believed to be an effective means of achieving the goals of low average access time and high store utilization. However, successful implementation of a system requires careful selection of design parameters concerning memory management.

AREAS OF EXTENSION AND FURTHER RESEARCH

The preceding sections dealt with the concept of failure-tolerant parallel programming and effective solutions to most fundamental problems involved in realizing its potential power. The model architecture devised for efficient execution of failure-tolerant parallel programs was specified at a highly abstract level in order to preserve simplicity and generality. It incorporated a minimal amount of facility. Naturally, the model architecture can be expanded in many directions to possess additional capabilities by incorporating various proven concepts and schemes. In addition, in order to put failure-tolerant parallel programming in general use, various program design and engineering tools need to be developed. Among numerous desirable extensions and research problems, only a few of the representative ones are listed below.

First, development of a language supporting failure-tolerant parallel programming is an immediate requirement. Such a language should contain more facilities for control structuring and data specification than the ones in conventional programming languages. No useful principles employed in other failure-tolerant program structuring schemes such as recovery block structuring,[11] need to be rejected in structuring failure-tolerant parallel programs. In view of the strong relationship between each object segment and the associated VR-segment, it is almost an indispensable requirement to put each pair of segments in a compartment in the program text. Such a compartment is called a *failure-tolerant segment*. More specifically, each VR-segment is directly dependent upon only one object segment in a failure-tolerant parallel program. Furthermore, the dependency structure among VR-segments is always the same as the dependency structure among object segments. The latter dictates the former. Thus the dependency among VR-segments as well as the dependency of a VR-segment on the associated object segment, need not be explicitly specified and should become a part of the definition of the failure-tolerant segment.

On the other hand, each object segment may be dependent on zero or more VR-segments which belong to other failure-tolerant segments. This type of dependency needs to be explicitly specified. This and other considerations mentioned in the previous section on desirable extensions of the state-of-art in failure-tolerant programming, led to the formulation of the model of the failure-tolerant segment depicted in Figure 7.

The model is a generalization of the model of the recovery block depicted in Figure 1. It consists of the *segment-head*, the primary object segment, several alternate object segments and the VR-segment. The segment-head specifies the prerequisite condition for entry into the (failure-tolerant) segment e.g., dependency on the completion of VR-segments belonging to other failure-tolerant segments. It specifies the *successor* failure-tolerant segments i.e. the ones whose initiation is dependent upon successful completion of the VR-segment. It also contains the declaration of the initial "execution priorities" of alternate object segments. It may also specify the types of abnormal conditions which may be recognized by the system during execution of object segments and the actions to be taken on occurrence of each condition e.g., "enter into the VR-segment," "record the occurrence and continue," etc. The VR-segment specifies the procedures of validation, VR-store management, and recovery including system reconfiguration and assignment of execution priorities to alternate object segments. The primary or alternate object segments can contain, nested within themselves, further failure-tolerant segments. However, the structuring rule illustrated in Figure 5 should become a part of the definition of the failure-tolerant segment.

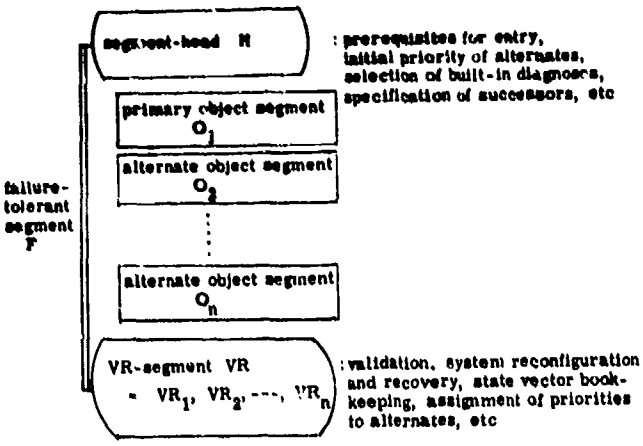

Figure 7—A structural model of a failure-tolerant segment

Therefore, programming of the segment-head requires some special language primitives including JOIN-like primitive" used for specifying the prerequisite condition for entry, FORK-like primitive" for specifying the successors, ones for specifying types of abnormal conditions and appropriate treatments, etc. Programming of the VR-segment also requires some special language primitives such as ones for referring to the old values of variables, ones for system reconfiguration, etc. Development of a language containing all the facilities mentioned above is urgent.

Second, if parallelism among object segments is to be exploited, the model architecture and the program structuring scheme described so far needs to be generalized accordingly. Such a generalization is expected to be a gigantic task requiring a great deal of research.

Third, it is often necessary to periodically save verified state vectors into the file store either as the spontaneous action of the system or as controlled by the

failure-tolerant program. Incorporation of an efficient filing capability into the model architecture is an essential requirement.

Fourth, in view of the high cost of a sizable content-addressable memory, it seems both necessary and desirable to use location-addressed modules as back-up memory when a certain program requires more space than that provided by available content-addressable modules. That is, additional content-addressable store space can be simulated on the basis of location-addressed modules and store structuring techniques such as hash-coding. Incorporation of the virtual memory into the model architecture will also be an interesting research subject.

CONCLUSION

The concept of failure-tolerant parallel programming was originated with the objective of utilizing extensive validation and recovery facilities at run-time without disturbing main-stream computation. The model architecture presented is believed to be a satisfactory solution to the efficient execution of failure-tolerant parallel programs. As further researches on the subjects mentioned in the preceding section progress, more insights will hopefully be gained into the potential power of failure-tolerant parallel programming and the cost-effective implementation of systems based on the model architecture.

ACKNOWLEDGMENT

This work was sponsored in part by the Joint Service Electronics Program under Air Force Contract F44620-71-C-0067 and in part by the National Science Foundation under Grant GJ-35833.

REFERENCES

1. Avizienis, A. et al., "The STAR (Self-testing and Repairing) Computers—An Investigation of the Theory and Practice of Fault-Tolerant Computer Design," *IEEE Trans. on Comp.*, November 1971, pp. 1312-1320.
2. Boyer, R. S. et al., "SELECT—A formal system for testing and debugging programs by symbolic execution," *Proc. 1975 Int'l Conf. on Reliable Software*, pp. 234-245.
3. Carter, W. C. and W. G. Bouricius, "A Survey of Fault-Tolerant Computer Architecture and its Evaluation," *Computer*, Jan.-Feb. 1971, pp. 9-16.
4. Chandy, K. M. and C. V. Ramamoorthy, "Rollback and Recovery Strategies for Computer Programs," *IEEE Trans. on Comp., February* 1972, pp. 137-146.
5. Chandy, K. M. et al., "Analytic Models for Rollback and Recovery Strategies in Data Base Systems," *IEEE Trans. on Software Engr.*, March 1975, pp. 100-110.
6. Chandy, K. M., "A Survey of Analytic Models of Rollback and Recovery Strategies," *Computer*, May 1975, pp. 40-47.
7. Chang, H. Y. et al., *Fault Diagnosis of Digital Systems*, Wiley-Interscience, 1970.
8. Connet, J. R. et al., "Software Defenses in Real-Time Control Systems," *Digest of the 1972 Int'l Symp. on Fault-Tolerant Computing*, pp. 94-99.
9. Conway, M., "A Multiprocessor System Design," *Proc. AFIPS Fall Joint Comp. Conf.*, pp. 139-146.
10. Dijkstra, E. W., "Structured Programming," in J. N. Buxton and B. Randell (eds.), *Software Engineering Techniques*, report on a Conf. sponsored by the NATO Science Committee, Rome, Italy, 1969, pp. 84-88.
11. Elemendorf, W. R., "Fault-Tolerant Programming," *Digest of the 1972 Int'l Symp. on Fault-Tolerant Computing*, pp. 79-83.
12. Elgpas, B. et al., "An Assessment of Techniques for Proving Program Correctness," *Computing Surveys*, June 1972, pp. 97-147.
13. Hetzel, W. C. (ed), *Program Test Methods*, Prentice-Hall, 1973.
14. Horning, J. J. et al., *A Program Structure for Error Detection and Recovery*, Lecture notes in *Comp. Sci.* Vol. 16, Springer-Verlag, 1974, pp. 177-193.
15. Kennedy, P. J. and T. M. Quinn, "Recovery Strategies in the No. 2 Electronic Switching System," *Digest of the 1972 Int'l Symp. on Fault-Tolerant Computing*, pp. 165-169.
16. King, J. C., "A New Approach to Program Testing" *Proc. 1975 Int'l Conf. on Reliable Software*, pp. 228-233.
17. Kopetz, H., "Software Redundancy in Real-Time Systems," *Proc. IFIP Congress 1971*, pp. 182-186.
18. Pierce, W. H., *Failure-Tolerant Computer Design*, Academic Press, 1965.
19. Pikner, H., "Programmed Restarts," *Proc. Annual ACM Conf.*, 1971, pp. 13-27.
20. Ramamoorthy, C. V., R. C. Cheung and K. H. Kim, "Reliability and Integrity of Large Computer Programs," Lecture notes in *Comp. Sci.*, Vol. 12, Springer-Verlag, 1974, pp. 86-161.
21. Ramamoorthy, C. V. and K. H. Kim, "Software Monitors Aiding Systematic Testing and their Optimal Placement," *Proc. 1st Nat'l Conf. on Software Engr.*, pp. 21-26.
22. Randell, B., "System Structure for Software Fault Tolerance," *IEEE Trans. on Software Engr.*, June 1975, pp. 220-232.
23. Rohr, J. A., "STAREX-Self-Repair Routines: Software Recovery in the JPL-STAR Computer," *Digest of the 1973 Int'l Symp. on Fault-Tolerant Computing*, pp. 11-16.
24. Rohr, J. A., *System Software for a Fault-Tolerant Digital Computer* Ph.D. thesis, Dept. of Comp. Sci., Univ. of Ill. at Urbana-Champaign, 1973.
25. Short, R. A., "The Attainment of Reliable Digital Systems Through the Use of Redundancy—A Survey," *IEEE Comp. Group News*, March 1968, pp. 2-17.
26. Von Neumann, J., "Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components," in *Automata Studies, Annals of Math.* No. 34, Princeton Univ. Press, 1956, pp. 43-98.
27. Yau, S. S. and R. C. Cheng, "Design of Self-Checking Software," *Proceedings 1975 International Conference on Reliable Software*, pp. 450-457.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: AFOSR-TR-76-0969	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): FAILURE-TOLERANT PARALLEL PROGRAMMING AND ITS SUPPORTING SYSTEM ARCHITECTURE	5. TYPE OF REPORT & PERIOD COVERED: INTERIM / 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): K. H. KIM and C. V. RAMAMOORTHY	8. CONTRACT OR GRANT NUMBER(s): F44620-71-C-0067, NSF-GJ-35833
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Electronic Sciences Laboratory, University of Southern California, Los Angeles, CA 90007	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: ~~61102F~~ ~~681306~~ 4751-01
11. CONTROLLING OFFICE NAME AND ADDRESS: The Joint Services Electronics Program, Technical Advisory Committee	12. REPORT DATE: 1976 / 13. NUMBER OF PAGES: 11
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): AF Office of Scientific Research (NE), Bolling AFB, Bldg. 410, Washington, D.C. 20332	15. SECURITY CLASS. (of this report): UNCLASSIFIED / 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

AFOSR-TR-76-0969

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

Conference paper: pp. 413-423, The National Computer Conference, 1976

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Computer software, failure-tolerant programming

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The state-of-art in software validation as well as the continuing growth of the size and complexity of software subsystems, makes extra costs paid for software error tolerance more than justified. A program is which software redundancy is incorporated i.3., a program in which procedures for run-time validation and recovery are explicitly specified, is generally called a failure-tolerant program. One problem in failure-tolerant programming, which could be particularly serious in real-time computing environments, is

next page

cont. → the program execution time increased due to incorporation of validation and recovery procedures. This paper introduces an approach to the solution, called the failure-tolerant parallel programming. The essence of this approach is to maximally overlap main-stream computation with redundant computation oriented for validation and recovery. Subsequently a model system architecture tailored for efficient execution of failure-tolerant parallel programs is described. It is of highly general and modular nature and contains a novel memory subsystem named the duplex memory. Directions of further researches on program structuring and expansion of the model architecture are also indicated.